Journal of Defense Modeling and Simulation: Applications, Methodology, Technology
XX(X) 1–17
© 2011 The Society for Modeling and Simulation International
DOI: 10.1177/1548512910395641
dms.sagepub.com

CADAC: Multi-use Architecture for Constructive Aerospace Simulations

Peter H Zipfel

Abstract

In today's network-centric world, aerospace vehicles interact with many objects. They navigate by overhead satellites, synchronize their flight paths with other vehicles, swarm over hostile territory and attack multiple targets. Studying these engagements with high-fidelity constructive simulations has become an important task of modeling and simulation (M&S). The simulation framework Computer Aided Design of Aerospace Concepts (CADAC) has its roots in FORTRAN code that dates back to the 1960s and was used by industry and the U.S. Air Force to simulate aerospace vehicles in all flight environments. To adapt CADAC to the new environment, a complete rewrite was carried out in C++, taking advantage of object-oriented programming techniques. The architecture of CADAC++ is based on the hierarchical structure of inherited classes. The vehicles (aircraft, missiles, satellites or ground targets), inherit the six-degree-of-freedom (6-DoF) equations of motion from the classes 'Flat6' or 'Round6', conveying either the flat or elliptical Earth model. In turn, these classes inherit the communication structure from the base class 'Cadac'. The components of the vehicle, e.g., aerodynamics, propulsion and autopilot, are represented by modules, which are member functions of the vehicle class. Communication among the modules occurs by protected module-variable arrays. Every instantiated vehicle object is encapsulated with its methods and data. To communicate between vehicles, data packets are loaded onto a global data bus for recall by other vehicles. Input occurs by ASCII file and output is compatible with CADAC Studio, a plotting and data processing package. CADAC++ is chiefly an engineering tool for refining the components of the primary vehicle and exploring its performance as it interacts (possibly repeatedly instantiated) with the multi-object environment. Its modular structure enables reuse of component models across simulations. In the 10 years of development, CADAC++ based constructive simulations have been built for many types of aerospace vehicles and integrated with mission-level simulations.

Keywords

aircraft and missile simulations, engagement simulations, high fidelity modeling, constructive simulations, C++ programming language, six degrees of freedom, CADAC, CADAC studio, three-stage booster, dual role missile, self defense missile, hypersonic cruise missile, multi-object programming, run-time polymorphism classes, communication bus.

1. Introduction

High-fidelity, six-degree-of-freedom (6-DoF) simulations play an important part in the development of weapon systems. These so-called constructive simulations are used in technology trade studies, preliminary design, hardware-in-the-loop evaluation, flight testing and training.[1]

The first all-digital, constructive simulations were created by the National Aeronautics and Space Administration (NASA), U.S. Department of Defense (DoD) and industry. In 1966 Litton Industries developed the architecture for a missile simulation in FORTRAN IV that had all of the features of a full 6-DoF simulation. It was the source of many derivatives by Hughes Aircraft, North American Aviation and Aerospace Corporation. Noteworthy is the U.S. Army ENDOSIM simulation.[a] The U.S. Air Force also adopted it to its own needs and named the simulation Computer Aided Design of Aerospace Concepts (CADAC).

a. AMTEC Corporation. *Endo-atmospheric Non-nuclear Kill Simulation*. Report No. TR 1147. Huntsville, AL: U.S. Army Strategic Defense Command, August 1989 (restricted distribution).

U.S. Air Force Research Laboratory, Eglin AFB, FL, USA

Corresponding author:
Peter H Zipfel, U.S. Air Force Research Laboratory, Eglin AFB, 73 Country Club Road, Shalimar, FL 32579, USA.
Email: mastech.zipfel@cox.net

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **FEB 2011**	2. REPORT TYPE	3. DATES COVERED **00-00-2011 to 00-00-2011**
4. TITLE AND SUBTITLE **CADAC: Multi-use Architecture For Constructive Aerospace Simulations**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **U.S. Air Force Research Laboratory,Eglin AFB, 73 Country Club Road,Shalimar,FL,32579**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology February 9, 2011.**		

14. ABSTRACT

In today?s network-centric world, aerospace vehicles interact with many objects. They navigate by overhead satellites, synchronize their flight paths with other vehicles, swarm over hostile territory and attack multiple targets. Studying these engagements with high-fidelity constructive simulations has become an important task of modeling and simulation (M&S). The simulation framework Computer Aided Design of Aerospace Concepts (CADAC) has its roots in FORTRAN code that dates back to the 1960s and was used by industry and the U.S. Air Force to simulate aerospace vehicles in all flight environments. To adapt CADAC to the new environment, a complete rewrite was carried out in C++, taking advantage of object-oriented programming techniques. The architecture of CADAC++ is based on the hierarchical structure of inherited classes. The vehicles (aircraft, missiles, satellites or ground targets), inherit the six-degree-of-freedom (6-DoF)equations of motion from the classes ?Flat6? or ?Round6?, conveying either the flat or elliptical Earth model. In turn, these classes inherit the communication structure from the base class ?Cadac?. The components of the vehicle, e.g., aerodynamics,propulsion and autopilot, are represented by modules, which are member functions of the vehicle class. Communication among the modules occurs by protected module-variable arrays. Every instantiated vehicle object is encapsulated with its methods and data. To communicate between vehicles, data packets are loaded onto a global data bus for recall by other vehicles. Input occurs by ASCII file and output is compatible with CADAC Studio, a plotting and data processing package. CADAC++ is chiefly an engineering tool for refining the components of the primary vehicle and exploring its performance as it interacts (possibly repeatedly instantiated) with the multi-object environment. Its modular structure enables reuse of component models across simulations. In the 10 years of development, CADAC++ based constructive simulations have been built for many types of aerospace vehicles and integrated with mission-level simulations.

15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**17**	

CADAC, since its inception in 1978, has morphed through many stages of improvements, but has remained faithful to its FORTRAN language. But in today's network-centric world, aerospace vehicles interact with many objects: they navigate by overhead satellites, synchronize their flight paths with other vehicles, swarm over hostile territory and attack multiple targets. Studying this connectivity has become an important aspect of high-fidelity simulations. FORTRAN, lacking the power of object-oriented programming, has therefore been replaced by C++. A new architecture, called CADAC++, was created to enable the conceptualization of aerospace vehicles.

Other organizations followed the same trend and converted from FORTRAN to C++, or started entirely new frameworks in C++. Best known is JSBSim,[2] an open-source aircraft simulation that is also the basis of the Flight Gear Simulator.[3] The U.S. Army created an entirely new framework called CMD C++ Model Developer.[4] At its core is a kernel that supports any kind of modeling described by time-phased differential equations. Its distribution is unrestricted. Another U.S. Army organization, MSIC (Missile and Space Intelligence Center) contracted with Dynetics for the MSIC++ Generic Simulation,[b] which is a multi-purpose missile simulation environment, but not generally available to the public.

CADAC is a joint development by the U.S. Air Force and the University of Florida. Its framework architecture and some of the academic simulations are publicly available. The original FORTRAN version and the plotting and analysis programs, CADAC Studio, can be downloaded from the American Institute of Aeronautics and Astronautics (AIAA).[5] The C++ simulations are available as a three-part Self-Study Series, based on lectures at the University of Florida.[6-8] CADAC Studio also supports these C++ simulations.

CADAC++, in its 10-year history, has been used as a simulation test bed for missiles, aircraft, unmanned aerial vehicles (UAVs) and spacecraft. Its modular structure enables reuse of subsystem models and its well defined interfaces allow integration into higher level simulations, like FLAMES®-based mission models.[9]

This paper summarizes the process that led from requirements definition to architecture development and full-up constructive simulation. Some examples are presented that highlight the features of CADAC++.

2. Requirements

CADAC++ is an engineering tool aiding in the development of aerospace vehicles. Although it focuses on the main vehicle – missile, aircraft, spacecraft – it also portrays the interactions with outside elements, such as satellites, targets and sister vehicles. The main vehicle is modeled with greatest fidelity, while the secondary objects have simpler representations.

The synthesis and conceptualization process places distinct requirements on the simulation architecture. To support the design engineer in evaluating the aerodynamics, propulsion, guidance and control components, CADAC++ should mirror the same modular structure and closely control the interfaces between them. It should encapsulate each vehicle object for multiple instantiation and provide global communication between them. Input and output must be flexible and compatible with CADAC Studio, a post-processing and analysis tool. More specific requirements follow.

2.1 Face to the User

Users like to focus on the evaluation of the main vehicle without being burdened by the details of the simulation's execution. They want control of the input/output and the vehicle modules that define the subsystems.

There should be only one input file that controls the simulation. It displays the run title, an option line for directing the output, the calling sequence of the modules, the sizing of the integration step and the initializing of the vehicle parameters. The integration step size should be variable. The aerodynamics and propulsion tables should be kept separate for safekeeping rather than being part of the source code. Their file names, given in the input file, would load the data decks into memory prior to execution. Multiple instantiation of the vehicle objects should be accomplished by simply duplicating the vehicle input data and changing selected variables as necessary.

The output control should be simple yes/no choices. An option line would provide output to the screen of the primary and secondary vehicles, together with the event messages that indicate their changing flight status. There should also be an option to archive the screen output to a file. Plot files as well as statistical data files would be written for individual vehicles and merged together for multi-vehicle displays. These output files should be compatible with the existing CADAC Studio for two- and three-dimensional plotting and statistical analysis.

The components of the vehicles should be mirrored by modules that model their features. Strict control of the interfaces will make the modules interchangeable amongst simulations. The modules should define these interface variables, execute integration of state variables and enable table look-up. Any vehicle changes that the user wants to make should be confined to these modules.

b. Dynetics. *MSIC++ Generic Simulation Documentation*. Report No.: MSIC C.M. Control Number COV0. Dynetics, 18 April 1995 (restricted distribution).

2.2 Multiple Encapsulated Vehicle Object

Each aerospace vehicle (be it missile, aircraft or spacecraft) should be built up from a hierarchy of classes, starting with the base class Cadac, followed by the equations of motion, and completed by the vehicle itself. Each vehicle is a C++ object with its data (aerodynamics and propulsion) and methods (modules) encapsulated. Run-time polymorphism should be used to sequence through the vehicle objects during execution.

2.3 Modularity of Vehicle Components

The modules, representing the vehicle components, should be public member functions of the vehicle classes. Their interfaces, the module-variables, would be stored in protected data arrays that are available to all modules of the vehicle object. During execution, the modules should define all module variables, make initializations, integrate state variables and conduct post-run calculations.

2.4 Event Scheduling

Just as aerospace vehicles transition though flight phases, the simulation should be able to sequence through such events. These events should be controlled by the input file without any code changes in the modules. Relational operators such as <, =, > would be applied to the module-variables and trigger the events.

2.5 Global Communication Bus

Because vehicle objects are encapsulated into classes, a global communication bus should enable the transfer of data. Each vehicle should be able to publish and subscribe to any of the module-variables.

2.6 Table Look-up

Table utilities should provide for one, two and three independent variable look-up. Tables must be stored in separate files and modifications easily accomplished. Simple syntax should make the table look-up easy to program in the modules.

2.7 Monte Carlo Capability

To automate the evaluation of random processes, a Monte Carlo methodology should be implemented. Distributions like uniform, Gaussian, Rayleigh, exponential and Markov should be identified in the input file by keywords. Stochastic output data must be written to files compatible with CADAC Studio for post-processing.

2.8 Matrix Utility Operations

The full power of C++ should be applied to matrix operations. Matrix utilities should be tailored to the specific needs of flight simulations and not burdened by C++ container classes. Efficient pointer arithmetic will speed up the execution and allow unlimited stringing of matrix operations.

2.9 Documentation and Error Checking

The module-variables, being the key interfaces between the modules, should be fully documented. The definitions provided in the modules should be collected in a single output file. The module-variables in the input file should also be documented with the same definitions.

Error checking should identify module-variables that have not been assigned the correct names or locations in the input file or the modules. Incompatible matrix operations should be flagged, as well as problems with opening of file streams. A variable should be displayed on the console that indicates the computational precision of the attitude calculations.

3. Architecture

These requirements can be satisfied with object oriented programming in C++. Hierarchical class structures, encapsulation of data and methods, run-time polymorphism, overloading of functions and operators, are all features used in CADAC++ to build a simulation environment suitable for flight vehicle synthesis.

CADAC++ programming follows the International Standard for C++ defined by the American National Standards Institute/International Organization for Standardization (ANSI/ISO) Committee in 1998 and implemented by most compilers like Microsoft Visual C++. Thus, portability is assured and low-cost operation is made possible.

Each requirement is now addressed separately, with particular focus on the classes that structure the features of CADAC++

CLASS	DESCRIPTION
Cadac,...	Hierarchical class structure of vehicles
Vehicle	Hosting a pointer array of type Cadac
Module	Storing module information
Variable	Declaring module-variables
Event	Storing event information
Packet	Declaring data packets for global communication bus

```
TITLE 2 Missiles (RF seeker) against 2 targets with 1 recce
MONTE 1 123456
OPTIONS y_scrn y_events y_tabout y_plot y_merge y_doc y_comscrn y_traj y_stat
MODULES
 Environment   def,exec
 kinematics    def,init,exec
 propulsion    def,init,exec
 aerodynamic   def,init,exec
 seeker        def,exec
 filter        def,exec
 ins           def,init,exec
 datalink      def,exec
 guidance      def,exec
 control       def,exec
 actuator      def,exec
 forces        def,exec
 euler         def,exec
 newton        def,init,exec
 intercept     def,exec
END
TIMING
 scrn_step 2
 plot_step 0.05
 traj_step 0.2
 int_step 0.001
 com_step 2
END
VEHICLES 5
 MISSILE6 Missile #1
 tgt_num 1 //'int' Target tail # attacked by 'this' missile module combus
 /Initial conditions
 sbel1 0 //Initial north comp of SBEL - m module newton
 sbel2 0 //Initial east comp of SBEL - m module newton
 sbel3 -4000 //Initial down comp of SBEL - m module newton
 dvbe 250 //Missile speed - m/s module newton
 /Aerodynamics
 AERO_DECK drmdr6_aero_deck.asc
 /Propulsion
 PROP_DECK drmdr6_prop_deck.asc
.........................................................................
 END
ENDTIME 18
STOP
```

Datadeck	Hosting a pointer array of type Table
Table	Storing tabular data
Markov	Storing Markov data
Matrix	Storing matrix operations
Document	Storing module-variable definitions

3.1 Face to the User

The user friendly requirements are met with an architecture that enables easy use and modification of the simulations. The input file has all the features that control the execution: title, option line, module call, timing control and vehicle initialization.
The option line provides nine possible outputs. During runtime, `y_scrn`, `y_event` and `y_comscrn` write data to the console; `y_tabout`, `y_plot`, `y_merge`, `y_doc`, `y_traj` and `y_stat` write the output to ASCII files for later processing by CADAC Studio. An important feature is the control that the user has over the loading and execution sequence of the modules. For sophisticated simulations, the calling sequence may become very important. Each of the timing events can be controlled separately in order not to overload the output devices. With the keyword `VEHICLES` begins the loading of the vehicle objects. Only a fraction of the first object `MISSILE6` is shown, though there are five vehicles to be loaded. By simply replicating `MISSILE6` or other objects and incrementing the integer after `VEHICLES`, new objects are loaded. Note how the file names that contain the tables are identified by the keywords `AERO_DECK` and `PROP_DECK`.

The user who wants to modify a vehicle component has only to deal with the corresponding module. The module contains all code and interfaces that define the components, carries out the table look-up and integrates the state variables. Re-use of modules for other simulations is facilitated by the strict control and detailed documentation of the interfaces.

As an example, let us look at the much abbreviated ‘newton’ module.

```
void Flat6::def_newton()
{
 //Definition and initialization of module-variables
flat6[210].init("VBEBD",0,0,0," Velocity deriv. - m/s^2","newton","state","");
flat6[213].init("VBEB",0,0,0,"Missile velocity - m/s","newton","state","");
flat6[230].init("FSPB",0,0,0,"Specific force - m/s^2","newton","out","");
flat6[239].init("hbe",0,"Height above ground - m","newton","out","scrn,plot");
flat6[247].init("mfreeze_newt","int",0,"Saving mfreeze ","newton","save","");
}
void Flat6::init_newton()
{
//initializations
...................
}
void Flat6::newton(double sim_time,double int_step)
{
//local module-variables
Matrix FSPB(3,1);
double hbe(0);
//localizing module-variables
//from initialization
...................
//getting saved value
int mfreeze_newt=flat6[247].integer();
//input from other modules
Matrix TBL=flat6[120].mat();
//state variables
Matrix VBEBD=flat6[210].vec();
Matrix VBEB=flat6[213].vec();
//------------------------------------------------------------------------
...................
//integrating acceleration in body coord to obtain velocity
FSPB=FAPB*(1/vmass);
Matrix VBEBD_NEW=FSPB-ATB+TBL*GRAVL;
VBEB=integrate(VBEBD_NEW,VBEBD,VBEB,int_step);
VBEBD=VBEBD_NEW;
...................
//------------------------------------------------------------------------
//loading module-variables
//state variables
flat6[210].gets_vec(VBEBD);
flat6[213].gets_vec(VBEB);
//saving values
flat6[247].gets(mfreeze_newt);
//output to other modules
flat6[230].gets_vec(FSPB);
flat6[239].gets(hbe);
//diagnostics
...................
}
```

The module consists of three parts: the definition of module variables, the initialization and the integration. In `def_newton()` the creation of capitalized `Matrix` variables, lower case `real` and `integer` variables is shown. Any new module-variable will be added here. Conversion of trajectory parameter from the input file to more suitable variables occurs in `init_newton()`. The integration takes place in `newton(…)` with a call to the function `integrate(…)`. This part shows the three sections of the code: creating or localizing variables, executing code and loading module-variables to the array `flat6[]`.

More detail of the modules is provided below under the heading Modularity of Vehicle Components.

3.2 Multiple Encapsulated Vehicle Object

The rewriting of CADAC was motivated by the unique feature of C++ allowing encapsulation of vehicle objects. Encapsulation means binding together data and functions while restricting their access. The aerodynamic and propulsion data are bound together with the table look-up functions and many other functions that support the missile and aircraft objects. In turn, these objects are created from a hierarchical class structure derived from the common *abstract* base class `Cadac`.

This hierarchical class structure in CADAC depends on the particular simulation. For instance, the CADAC 6-DoF aircraft simulation consists of a single branch `Cadac` ← `Flat6` ← `Plane`, where `Flat6` models the equations of motion over the flat Earth, and `Plane` models the components of an airplane. The more elaborate CADAC missile engagement simulation has multiple branches. Its main branch represents the high-fidelity 6-DoF missile model `Cadac` ← `Flat6` ← `Missile`. The supporting vehicle branches `Cadac` ← `Flat3` ← `Target` and `Cadac` ← `Flat3` ← `Recce` are the 3-DoF target and reconnaissance aircraft. As another example, the 3-DoF CADAC cruise missile simulations over the round rotating Earth has the three branches: `Cadac` ← `Round3` ← `Cruise`, `Cadac` ← `Round3` ← `Target` and `Cadac` ← `Round3` ← `Satellite`.

The vehicle objects, declared by their respective classes, are created during run-time by the polymorphism capability of C++. Polymorphism (many forms, one interface) uses inheritance and virtual functions to build one vehicle-list of all vehicle objects, be they 6-DoF missiles, 3-DoF targets and recce aircrafts or satellites. At execution, this vehicle-list is cycled through at each integration step in order to compute the respective vehicle parameters.

The class **`Vehicle`** facilitates the run-time polymorphism. It has a private member `**vehicle_ptr`, which is a pointer to an array of pointers of the class `Cadac` that contains the pointers to all the vehicles objects. It also declares the offset operator `Cadac *operator[](int slot)` that returns the pointer to the vehicle object located at the offset `slot` in the vehicle-list.

In `main()`, the object `Vehicle` **`vehicle_list`**`(num_vehicles)` is created, initialized and its constructor allocates memory for the array of pointers

```
vehicle_ptr=new Cadac *[num_
        vehicles];
```

Then the global function `Cadac *`**`set_obj_type`**`(...)` interrogates the input file `input.asc` to identify the vehicle types by keywords such as MISSILE6, TARGET3, RECCE3, etc. It allocates memory to the vehicle objects `Missile, Target, Recce`, etc. and returns pointers of base class `Cadac.` These pointers are stored in the `vehicle_ptr[]` array by the `Vehicle` member function

```
vehicle_list.add_vehicle
    (*vehicle_type);
```

Now the `Vehicle` object `vehicle_list` is ready to be addressed by its offset operator `[]`. For instance, the vehicle specific data are read from `input.asc` by

```
vehicle_list[i]->vehicle_
        data(input);
```

where `i` is the `(i+1)`th vehicle object in the sequence established in `input.asc`. Here is the explanation of the logic flow. The offset operator `[]` takes `i` and returns the `vehicle_ptr[i]` of the `(i+1)`th vehicle. Although the `vehicle_ptr` array is of the base class `Cadac`, the compiler has knowledge of the individual pointer being of the derived class `Missile, Target` or `Recce`. Such is the marvel of run-time polymorphism! Another important example is the call of a vehicle module, say the aerodynamic module

```
vehicle_list[i]->aerodynamics();
```

If the `(i+1)`th vehicle is the MISSILE6, a pointer of type Missile is furnished that is used to call the member function `aerodynamics()` of the derived class `Missile`. On the other hand, if the vehicle is the TARGET3, the pointer is of type `Target` and points to the `Target` member function `aerodynamics()`.

Through run-time polymorphism any number of different vehicles can be called using the common pointer array of type `Cadac`. These calls are executed during initialization and at every integration step. A limitation of this architecture is that all vehicle objects have to be instantiated at the beginning of the run.

3.3 Modularity of Vehicle Components

A key feature of CADAC is its modularity, which reflects the component structure of an aerospace vehicle. Just as the hardware is divided into subsystems (such as propulsion, autopilot, guidance and control) CADAC simulations are broken into propulsion module, autopilot module, etc. This is extended to include non-hardware modules like aerodynamics, Newton's and Euler's equations of motion and environmental modules. This one-for-one correspondence ensures clean interfaces between the modules.

Each module is a pure virtual member function of the abstract base class `Cadac` and is overridden in the derived class, be it `Flat6, Flat3, Missile, Target, Recce` or others. If the derived class does not use a module, the module will return empty.

The calling sequence of the modules is controlled by their sequential listing in the input file `input.asc`. Each module may consist of four parts: the definition part (identified by `def`), the initialization part (`init`), the execution part (`exec`) and the last call (`term`). All are called only once, with the exception of `exec` which is called during every integration step.

The structure **`Module`** declares the name and the four parts of the module. Reading from `input.asc`, the modules are loaded into the `module_list` by the global function

```
order_modules(input,num_
    modules,module_list);
```

At creation of the vehicle object, at module initialization and at each integration cycle the `module_list` is interrogated for the module names. For example, the definition of the aerodynamic module occurs in the vehicle's **constructor**

```
if((module_list[j].
name=="aerodynamics")&&(module_list[j].
definition=="def"));

def_aerodynamics();
```

Then the initialization of the module takes place in **`main()`**

```
if((module_list[j].
name=="aerodynamics")&&(module_list[j].
initialization=="init"))

 vehicle_list[i]->init_aerodynamics();
```

where `vehicle_list[i]` is the pointer to the vehicle object. During integration, the module is called inside the **`execute`**`(...)` function which is called directly from `main()`

```
            if(module_list[j].
           name==”aerodynamics”)
    vehicle_list[i]->aerodynamics();
```

Currently, the terminal calls are not needed.

Data are transferred between modules by **module-variables** and stored in arrays of type `Variable`. Each derived object from the base class `Cadac` has an array with its own name, such as `flat6[], missile[], target[]`, etc. They are protected members of `Cadac`. The arrays are sized by global constants `NFLAT6, NMISSILE, NTARGET`, etc. and each module is assigned a block of indices in its respective arrays.

The class **`Variable`** declares the module-variable object. Its private members store the label, the initial value, the type of variable (`int`, `double`, 3 x 1 vector, 3 x 3 matrix), the definition and units, the module where the value is calculated, its role (input data, state variable to be integrated, diagnostic, output to other modules and data saved for the next integration cycle), the output direction (screen, plot file, communication bus) and two error codes. The public methods of `Variable` contain a four times overloaded function `init(…)` for integer, double, vector and matrix variables which are used for the variable definitions in the definition part of the module, e.g.

```
missile[110].init(“ca”,0,”Axial
force coefficient”,”aerodynamics”,
”out”,”plot”);
```

Other public methods of Variable govern the reading and loading of the module-variables inside a module. To make the module-variables local variables, the member functions `integer(), real(), vec()` and `mat()` are used. For instance,

```
        int mfreeze_newt=flat6[247].
        integer();
        double grav=flat6[55].real();
        Matrix TBL=flat6[120].mat();
        Matrix FAPB=flat6[200].vec();
```

By convention, scalar variables are named with all lowercase letters, while upper-case letters designate matrices. Only 3 x 1 vectors and 3 x 3 matrices are permitted as module-variables.

The loading of the local module-variables into the protected arrays uses the member functions `gets(…)`, `gets_vec(…)` and `gets_mat(…)` where `gets(…)` is overloaded and serves both `int` and `double` types. For instance,

```
        flat6[247].gets(mfreeze_newt);
        flat6[248].gets(dvbef);
        flat6[230].gets_vec(FSPB);
        flat6[120].gets_mat(TBL);
```

Module-variables provide the sole data transfer between the modules of a vehicle object. For documentation they are recorded in sequential order in `doc.asc` with their definitions and other relevant information. Between their label and array location, there is a unique one-to-one relationship. Any deviation from that rule is flagged in `doc.asc`.

3.4 Event Scheduling

As aerospace vehicles fly their trajectories, they may sequence through several events towards their destinations. Just think of rockets staging, airplanes taking off, cruising and landing and missiles passing through midcourse and terminal phases towards the intercept. Events in CADAC++ are interruptions of the trajectory for the purpose of reading new values of module-variables. They can only be scheduled for the main vehicle object. The maximum number of events is determined by the global integer `NEVENT`, while the number of new module-variables in each event is limited by the global integer `NVAR`.

An event is defined in the input file `input.asc` by the event block starting and ending with the keywords `IF … ENDIF`. Appended to `IF` is the event criterion. It consists of the watch variable (any module-variable except of type Matrix) and a relational operator followed by a numerical value. For instance,

```
IF dbt < 8000
 mseek 12 //’int’ =x2:Enable,
=x3:Acquisition, =x4:Lock module seeker
ENDIF
```

means, if the range to the target is less than 8000 m, the seeker is enabled. The supported relational operators are `<, =, >`.

The **`Event`** class supports the creation of `Event` type objects. The pointer of each event is stored in the **`event_ptr_list`**`[NEVENT]`, which is a protected member of the vehicle class. The private members of the `Event` class store information about the event, such as watch variable, relational operator, threshold value and new module-variables. The public methods are ‘set’ and ‘get’ functions for the data. To expedite execution, the new module-variables are not stored by their name, but by their offset index in the module-variable array. Therefore, rather than cycling through all the module-variables, the new module-variables are directly picked out by their offset indices. These index lists are also part of the private data members of `Event`.

Event data are read in `main()` from `input.asc` by the vehicle member function

```
vehicle_list[i]->vehicle_data(input);
```

for each vehicle object and they are ’set‘ into `Event` objects, whose pointers are stored in the `event_ptr_list`.

In the function `execute(…)`, the watch variables are monitored at every integration interval by the vehicle member function **`event(…)`**. If the criterion of an event is satisfied, the new values for the module-variables are loaded and a message is written to the console to announce the event.

Event scheduling provides great flexibility to shaping the trajectory of an aerospace vehicle. However, as a design matures and the switching logic becomes well defined, the events can be scheduled in the module itself and any event scheduling in the `input.asc` file may be completely eliminated at the inconvenience of having to recompile the module if changes are made.

3.5 Global Communication Bus

Encapsulation by classes isolates vehicle objects from each other. However, this feature of C++ prevents direct communication between the vehicles. For instance, the missile object needs to know the coordinates of the target object in order for its seeker to track it. How can the missile get access to the protected target data?

In CADAC++ the global communication bus, called **`combus`**, provides this interface. Selected module-variables are stored in `combus` so that other vehicles can download them. To identify this process we use the terms ‘**publish**’ and ‘**subscribe**’.

Every vehicle prepares a data set of module-variables and publishes it to `combus`. These module-variables are identified by the keyword ‘`com`’ in their definition; for instance `vmach` is added to the data set by

```
flat6[56].init(“vmach”,0,”Mach
 number”,”environment”,”out”,
       ”scrn,plot,com”);
```

Any vehicle can subscribe to the data set of any other vehicle. Utility methods enable the process.

The enabling global class is **`Packet`**. One of its private data member stores the vehicle ID, the status of the vehicle (alive, hit, dead), the number of module-variables in the data set and a pointer to the array of module-variables of type `Variable`. Each vehicle object contributes one packet to the communication array `combus` of type `Packet`. The slot # is the same as that of the vehicle in the `vehicle_list`.

Under the `main()` scope, the pointer to the communication array is created (`Packet  *combus`) and dynamic memory is allocated (`combus=new Packet [num_vehicles]`), where the array is dimensioned by the number of vehicle objects. In the vehicle constructor, the vehicle member function `com_index_arrays()` is called, which collects the offset indices of the module-variables into integer arrays. Still under the `main()` scope, `combus` is initialized with the packet of each vehicle i

```
combus[i]=vehicle_list[i]
->loading_packet_init(…);
```

Then, in function `execute(…)`, at every integration step, the values of the module-variables are updated

```
combus[i]=vehicle_list[i]->loading_
           packet(…);
```

Each packet has a data set of module variables stored in the array pointed to by Variable `*data`. The storage sequence in the data set is determined by the order the module-variables are read. The module sequence is defined in the input file `input.asc`. This sequence is important for the subscription process.

The subscription of module-variables occurs in the modules. For instance, the seeker in order to track the target has to subscribe to the target position and velocity. First, the target ID is built from the string ’t‘ and the tail number of the target. Then `combus` is searched for this packet and the data set is downloaded

```
data_t=combus[i].get_data();
```

Knowing that the target position and velocity vectors are at offset 1 and 2, they can be subscribed

```
Matrix STEL=data_t[1].vec();
Matrix VTEL=data_t[2].vec();
```

Now `STEL` and `VTEL` of the target object are local variables of the missile object and can be used by the seeker.

The number of module-variables in the data set is unrestricted. If you are unsure of the storage sequence, you can find it by selecting `y_comscrn` and counting the labels. However, be aware that the three components of vectors count as one label only.

3.6 Table Look-up

Interpolating aerodynamic and propulsion tables is an important task in any aerospace simulation. Aerodynamic coefficients are usually given as functions of incidence angles and Mach number. Sometimes they are also expressed as functions of altitude and control surface deflections. Propulsion data are tailored to the type of propulsion system. For rocket motors, simple thrust tables may suffice. Turbojet and ramjet engines depend on throttle, Mach number and sometimes on incidence angles.

The more variables are used to describe the system, the greater the complexity of the table. Seldom is the dimension higher than three due to run-time considerations. CADAC++ supports table look-up schemes up to third dimension and interpolates linearly between the discrete table entries. It keeps the so-called 'data decks' as separate files so they can be properly protected as the need may arise.

The handling of the tables is accomplished by two classes, `Datadeck` and `Table`. The class **`Datadeck`** has a private member `**table_ptr`, which is a pointer to an array of pointers of the class `Table` that contains the pointers to all of the tables of a data deck. Under the 'main vehicle' scope, inside the 'protected' access specifier, the objects `Datadeck aerotable` and `Datadeck proptable` are declared along with the table pointer `Table *table`. At execution, two distinct phases take place: loading the tables and extracting the interpolated value.

The loading of the tables starts when the file `input.asc` is read by the function

```
    void input_data(fstream &input);
```

and the keywords `AERO_DECK` and `PROP_DECK` are encountered with their trailing file names. Then the calls

`read_tables(file_name,aerotable)` **and**
`read_tables(file_name,proptable);`

execute the code of the function

```
    void read_tables(char *file_name,
          Datadeck &datatable);
```

which picks up one of the `file_name` and returns by reference the object `datatable` of type `Datadeck`. Internally, `read_tables(…)` opens the data file and allocates dynamic memory first to the array of `Table` pointers (pointed to by `**table_ptr` which is a private member of the object `Datadeck aerotable`) then to the `Table` object (pointed to by `table`) and its data arrays. Now the numerical values are read into the data arrays for each table and `read_tables(…)` returns void. Both functions `input_data(…)` and `read_tables(…)` operate within the scope of the 'main vehicle' object.

The extraction of the interpolated value occurs in the modules. The `Datadeck` objects `aerotable` or `proptable`, declared under the 'main vehicle' object, give access to the public `Datadeck` member function

```
          double look_up(string name,
               double value1,…);
```

which is overloaded three times for one, two and three independent variables. A typical example, taken from an 'aerodynamic' module of a two-dimensional table look-up, is

```
    double cm=aerotable.look_up("cm_vs_
         elev_alpha",delex,alphax);
```

It returns the interpolated value. This call to `look_up(…)` initiates two other calls to member functions of `Datadeck`. First, in **`find_index`**`(…)`, a binary search locates the indices of the independent variables just below the table entries. Then **`interpolate`**`(…)` linearly interpolates between the next higher discrete value and passes the interpolated value back up to the `look_up(…)` function for return.

Additions and deletions of tables in the AERO_DECK or PROP_DECK are automatically adjusted during the loading of the tables. If a simulation requires data tables of a different type (e.g., antenna pattern) one has to do four things: (1) create an ASCII file with the data tables, (2) identify the file name by a keyword `ANT_DECK antenna_data.asc` in the `input.asc` file, (3) declare an additional `Datadeck` object in the 'main vehicle' class `antennatable` and (4) replicate in the function `input_data(…)`.

3.7 Monte Carlo Capability

High fidelity simulations use random variables to model noise, disturbances and uncertain phenomena. If we make a single run, it represents only one realization of the total population of random variables. To do a complete stochastic analysis, many repetitive runs have to be executed, each drawing a different value from a distribution. This process can be automated and is referred to as the Monte Carlo capability of a simulation.

Randomized variables may be needed at initialization (e.g., seeker bias) or during the execution of the simulation (e.g., seeker noise). CADAC++ supports both. They are identified in the input file `input.asc` by the capitalized keywords that designate their distributions `UNI, GAUSS, RAYL, EXP` and `MARKOV`. The first four are used for initialization only. `MARKOV` models a Markov process with Gaussian distribution and first-order time correlation. It has to be called every integration cycle.

To initiate a Monte Carlo run, the keyword `MONTE` with two arguments is inserted right before the `OPTION` line in `input.asc`. The first argument is the run repetition number and the second is the random number seed. If the repetition number is set to zero, one run is executed using the mean values of the distributions.

The stochastic variables, identified in `input.asc`, are read by the vehicle object function

```
        vehicle_data(fstream &input,int
                   nmonte);
```

If there are initialization variables, a value is drawn from their respective distribution and held constant until it is re-initialized for the next trajectory. Module-variables identified by `MARKOV` are initialized with their Gaussian distribution and stored in the `markov_list`, which is of type `Markov` and sized by the global integer `NMARKOV.` If `nmonte=0,` the mean values are selected.

The class **`Markov`** handles the storage of the Markov data. It declares as private members the sigma and time correlation values of the Markov process and the index of the module-variable of its array. The `Markov markov_list` is a protected member of the `Cadac` hierarchy. Therefore, each main vehicle object has its own list of Markov variables with its own random draws.

At every integration step, for each vehicle, the Markov noise function is called in function `execute(...)`

```
vehicle_list[i]->markov_noise(sim_
time,int_step,nmonte);
```

This function downloads the Markov data from the `markov_list` and calls the utility function **`markov`**(...) to refresh the value.

Because Markov noise is a first order correlation process, the current value depends on the previous value. Therefore, there is a provision in the `Markov` class to save the current value for the next cycle.

Stochastic analysis is an important aspect of the performance evaluation of any aerospace vehicle. CADAC++ support stochastic initialization for all vehicles, but reserves the Markov process for only the main vehicle. For post-run analysis, the stochastic data of the main vehicle are written to `stat` files by exercising the `OPTION` **`y_stat`**.

3.8 Matrix Utility Operations

Modern programming uses matrix operations wherever possible to condense code and eliminate errors caused by coordinating equations. CADAC++ has a rich set of matrix operations which are public members of the class **`Matrix`**. This class is tailored to the special needs of flight dynamics. Generality has been sacrificed for efficiency. Rather than using template classes and particularly the vector container class of the STL, the CADAC++ matrix operations are restricted to variables of type `double`.

The class **`Matrix`** declares a private pointer to the matrix array double **`*pbody`** together with the array dimensions. There are 55 matrix operations declared in the public access area. They are divided into 34 functions and 21 overloaded operators.

In the following examples, capitalized variables are arrays, lower-case names are either functions or scalars.

```
Matrix AAPNB=TBLC*WOELC.skew_
sym()*UTBLC*gnav_mid_pn*dvtbc;
```

This example calculates the (3 x 1) acceleration vector `AAPNB` from the LOS rates `WOELC.`

The next example calculates the 8 x 8 gain matrix of the filter:

```
Matrix GK=PMAT*~HH*INV.
inverse();
```

And, finally, the 3 x 1 accelerometer vector error is determined by

```
Matrix EAB=ESCALA.diamat_
vec()+EMISA.skew_sym();
```

Note the limitless possibilities of stringing together matrix operations.

Matrix variables are created by specifying their name and dimensions, e.g., `Matrix MAT(3,6)`. The constructor allocates dynamic memory to the `Matrix` pointer `*pbody` and zeros all elements. The operations themselves use `pbody` and perform pointer arithmetic to accomplish the various matrix manipulations. Those operations that recreate a matrix `return` **`*this`**; i.e. they return the re-calculated object that was originally created and initialized, e.g.

```
Matrix UNIT(3,3);

UNIT.identity();
```

The matrix utilities have a full suite of overloaded operators. The assignment operator requires a **copy constructor** to provide for a deep copy of the object to assure that the new object has its own memory allocated and that it is recoverable when the object is destroyed.

The offset operator **`[]`** is also overloaded to access the elements of a `Matrix` array. However, this works only for one-dimensional arrays because two-dimensional arrays require more than one offset operator. For those instances, the Matrix functions `assign_loc(..)` and `get_loc(...)` must be used.

3.9 Documentation and Error Checking

Self-documentation is an essential part of any simulation. Of primary interest are the variables that are used for input/output as interfaces between modules and those of particular interest for diagnostics. All are referred to as **module-variables**. The description of a module-variable occurs only once in the 'def_module' function. This description is used to document the input file `input.asc` and to create a list of all module-variables in the output file `doc.asc`. The documentation of `input.asc` is automatic, while the file `doc.asc` is only created if the `OPTION y_doc` is selected.

CADAC error checking focuses in particular on the correct formatting of the `input.asc` file and the enforcement of the one-to-one correspondence rule, 'One module-variable name for one array location'. Other checks assure that matrix operations are performed on compatible matrices and that file streams open correctly.

The class **`Document`** is used to make the module-variable descriptions available. Its private data are essentially a subset of the class `Variable`. They store name, type, definition and module of each module-variable. Under the `main()` scope (during initialization), arrays of type `Document` are built for each vehicle object, followed by the function call `document()`

```
        vehicle_list[i]->document
        (fdoc,title,doc_missile6);
```

This function, under the vehicle object scope, writes to `doc.asc` formatted information of each module-variable and identifies the empty slots in the module-variable arrays. Under the same scope, `input.asc` is documented if `OPTION y_doc` is set

```
    document_input(doc_missile6, doc_
           target3,doc_recce3);
```

This function operates under the global scope. It uses the arrays of type `Document` to extract the module-variable descriptions and appends them after the numerical value of the variable. If it cannot find a matching name it prints out an error message.

To flag violations of the one-on-one correspondence rule, both `Document` and `Variable` classes cooperate. In the `Variable` class the private `char error[2]` holds the error codes (* or A).

During initialization, as the **`init`**`(…)` functions of the modules are called, a check is made whether that slot is empty and can receive a new variable. If not, the error code '*' is set. As `document (…)` writes the output file `doc.asc,` the module-variable array is checked for duplicate names. The error code 'A' is set if this occurs. Both codes are inserted in the first column of the `doc.asc` file and a warning message is sent to the console.

A good description of a particular simulation is produced if the modules, the `input.asc` and the `doc.asc` files are combined in a document. It should enable someone else, who is familiar with the CADAC++ architecture, to pick up, run and understand the simulation.

4. Constructive Simulations

Constructive simulations have become the engineer's major integration tool. With their realistic portrayal of the physical interactions between aerodynamics, propulsion, guidance and control they support concept studies, hardware integration tasks, flight testing and training. Specifically they enable the following:

- *Developing performance requirements.* A variety of concepts are simulated to match up technologies with requirements and to define preliminary performance specifications.
- *Performing technology trade studies.* Various subsystem are modeled and analyzed as they interact with other subsystems to determine the specifications that best meet the performance requirements.
- *Guiding and validating designs.* Sensitivity studies are conducted to determine the optimal design parameters; models of subcomponents are tested as they interact with other parts of the system; overall performance is established.
- *Test support.* Test trajectories and footprints are pre-calculated and test results are correlated with simulations.
- *Reduction in test cost.* A simulation, validated by flight test, is used to investigate other points in the flight envelope.
- *Investigating inaccessible environments.* Simulations are the most cost effective way to check out vehicles that fly through the Martian atmosphere or land on Venus.
- *Pilot and operator training.* Thousands of flight simulators help train military and civilian pilots.
- *Practicing dangerous procedures.* System failures, abort procedures, and extreme flight conditions can be explored safely on simulators.
- *Gaining insight into flight dynamics.* Dynamic variables can be traced through the simulation and limiting constraints can be identified.
- *Integration of components.* Understanding how subsystems interact to form a functioning vehicle.

CADAC, during its long history, has supported all of these tasks. It has been used to develop missile performance requirements, to conduct technology trades (airframe, propulsion, seeker, guidance and control), and to support flight test planning and data analysis. Air-to-air missile concepts have been integrated into air combat domes and into mission-level simulation frameworks like FLAMES®.

Not all CADAC simulations are at the 6-DoF fidelity, though high-fidelity modeling is required for delivery accuracy, hardware-in-the-loop and flight testing. However, during weapon conceptualization, lower fidelity 5-DoF or 3-DoF models often suffice or are mandated by the lack of detailed component data. These simpler simulations drop one or three of the attitude degrees of freedom. Most of the recent CADAC simulations are of 6-DoF fidelity, but some UAV, air-to-air and air-to-ground missiles are modeled at 5-DoF. All these simulations have the same CADAC++ framework. The distinction is reflected in the vehicle class structure and the associated modules.

Table 1. CADAC++ Active Simulations

TYPE	VEHICLE OBJECTS	DoF	EARTH	FEATURES
Cruise Missile	Missile; Target; Satellite	5	Spherical	Remote Targeting
Fighter Aircraft	Aircraft	6	Flat	Generic F16
Air-to-Ground Missile	Missile; Aircraft; Target	6	Flat	Weather Deck, MC
Air-to-Air Missile	Missile; Target Aircraft	6	Flat	MC
National Aerospace Plane	Plane + Transfer Vehicle + Interceptor; Tracking Station; Satellite	6	WGS84	Generic X30, Weather Deck, MC
Generic Defense Missile	Defensive Missile; Aircraft; Offensive Missile	6	Flat	MC
Three Stage Booster	Rocket with Three Stages	6	WGS84	Insertion Guidance, Weather Deck, MC
Long Range Strike Missile	Missile; Target	5	Spherical	Hypersonic, FLAMES®
Dual Role Missile	Missile; Target; Recce Aircraft	6	Flat	Two Pulse Rocket, Integral Rocket Ramjet, MC
Global Strike	Booster + Waverider + Munition; Satellite; Target	6	WGS84	Wave Rider, MC
Self Defense Missile	Defensive Missile; Aircraft; Offensive Missile	6	Flat	Real time, MC, FLAMES®
Small Smart Bomb	Bomb; Satellite; Target	6	WGS84	Weather Deck, MC
Hypersonic Cruise Missile	Missile; Satellite; Target	6	WGS84	Scramjet, Weather Deck, MC

MC=Monte Carlo capable

For efficient use of constructive simulations, versatile plotting options and stochastic data processing must be available. CADAC Studio satisfies that need. Its history is as distinguished as that of the CADAC simulations. Originally developed for mainframes, then for VAXes and finally for PCs, it provides for interactive plotting, automated launch envelope and footprint generation and post-processing of Monte Carlo runs.

The plotting options consist of two dimensional traces and three dimensional trajectories either in a Cartesian grid or in longitude, latitude, altitude over the globe. A strip-chart capability plots up to 12 traces against time.

The SWEEP program of CADAC Studio automates the generation of footprints for air-to-ground missile and launch envelopes for air-to-air missiles. A single CADAC submittal will spawn trajectory runs against a target grid. Plotting options then generate carpet plots of selected parameters.

Monte Carlo runs draw from a variety of stochastic distributions for insertion of numerical values into uncertain parameters such as aerodynamic uncertainties, INS errors, seeker errors and wind gusts. CADAC Studio analyzes the output and generates CEPs, bivariate ellipses and mean values in the target or intercept planes with plots that also display the scatter points. Even the SWEEP program can be executed in the Monte Carlo mode to generate CEPs throughout the envelope or footprint.

CADAC and CADAC Studio provide a complete environment for constructive simulations. They have been used for bombs, missiles, UAVs, aircraft and spacecraft. Some recently developed simulations are summarized next.

5. Multi-use Examples

During the course of the CADAC++ development many simulations were built on the same framework. The modular architecture lends itself to multi-use applications. For example the environmental module is suitable for all simulations. The 6-DoF equations of motion over the WGS 84[10] or flat Earth apply to all corresponding simulations (Newton and Euler modules are the same). Seeker, INS, guidance and control laws can be shared where appropriate. With the strict enforcement of the interfaces between modules, the integration of an existing module into a new simulation is simplified.

Table 1 summarizes the currently active CADAC++ simulations. They cover the breadth of aerospace vehicles. Among the 13 simulations are two 5-DoF models, six over the flat Earth, and five over the elliptical WGS84 Earth. Most have Monte Carlo capability and use the U.S. Standard Atmosphere 1976;[11] some can implement a test atmosphere with winds aloft as well as Dryden-type turbulence.[12] The simulations are also distinguished by the type of vehicle objects they model. Some have just one object, while others have two or three. The first object is always the vehicle of primary interest. It determines the degree-of-freedom classification. It can morph into different configurations as indicated by the '+' sign. The semicolon separates the objects. Each vehicle object can be instantiated multiple times making possible the engagement of many-on-many. Two simulations have been integrated into the mission-level FLAMES® framework: Long Range Strike Missile and Self Defense Missile.

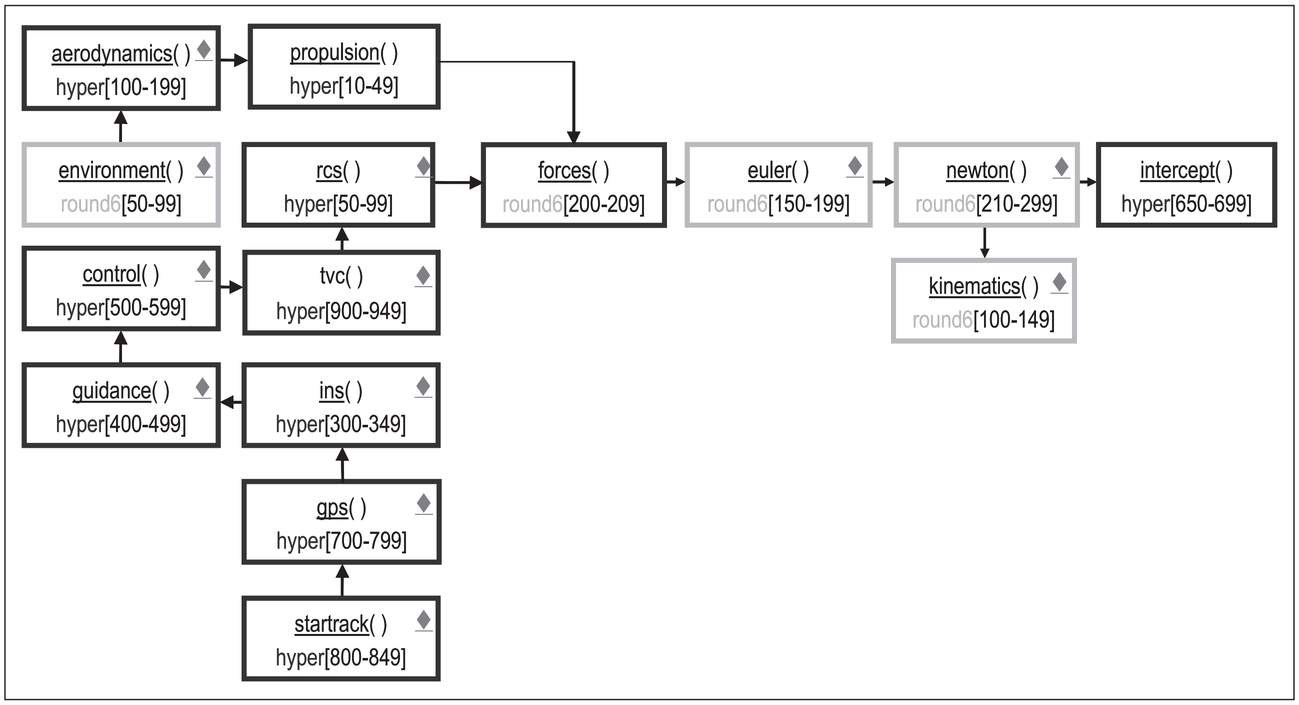

Figure 1. Modular architecture of the three stage booster simulation.

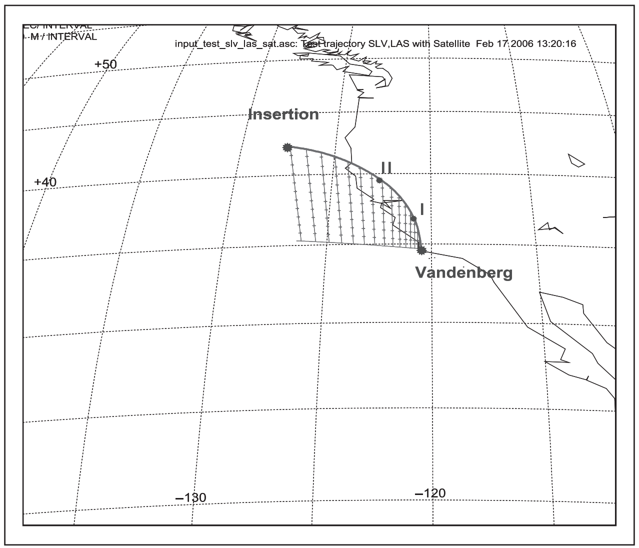
input_test_slv_las_sat.asc: Test trajectory SLV,LAS with Satellite Feb 17 2006 13:20:16
+50
+40
Insertion
II
I
Vandenberg
−130
−120

Figure 2. Ascent of the three stage booster and suborbital insertion.

Two simulations are presented as examples. The Three Stage Booster simulation represents a model of a rocket that can place a payload into low Earth orbit using the WGS84 equations of motions. The Generic Defense Missile simulation (which models three objects) represents a blue missile launched from a blue aircraft against an incoming red missile using the flat Earth equations of motions.

5.1 Three Stage Booster Simulation

This is a typical solid rocket delivery booster. It is controlled by thrust vector control (TVC) and reaction control system (RCS) but has no aerodynamic control fins. The autopilot uses accelerometer and rate gyro feedback from the inertial measurement unit (IMU) to steer the missile. During the first stage, a pitch program is executed while maintaining small incidence angles in the high dynamic pressure region. Stages two and three are under ascent guidance to meet the terminal insertion conditions. This guidance law implements linear tangent guidance for minimum fuel consumption.[13,14] The onboard inertial navigation system (INS), updated by the global precision system (GPS) and a star tracker, provides the navigation states of the booster.

The class hierarchy of this simulation has only one branch, Cadac ← Round6 ← Hyper. ‘Round6’ models the 6-DoF equations of motion over the WGS84 Earth and ‘Hyper’ contains all the subsystems of the booster coded in modules (see Figure 1). The protected arrays of the classes are labeled `round6[]` and `hyper[]`and the assigned locations are indicated in the brackets.

A typical trajectory is launched and places a payload at the suborbital conditions of 110 km altitude, 1.5° flight path angle and 6600 m/s inertial speed. Figure 2 was generated with the CADAC Studio Globe program.

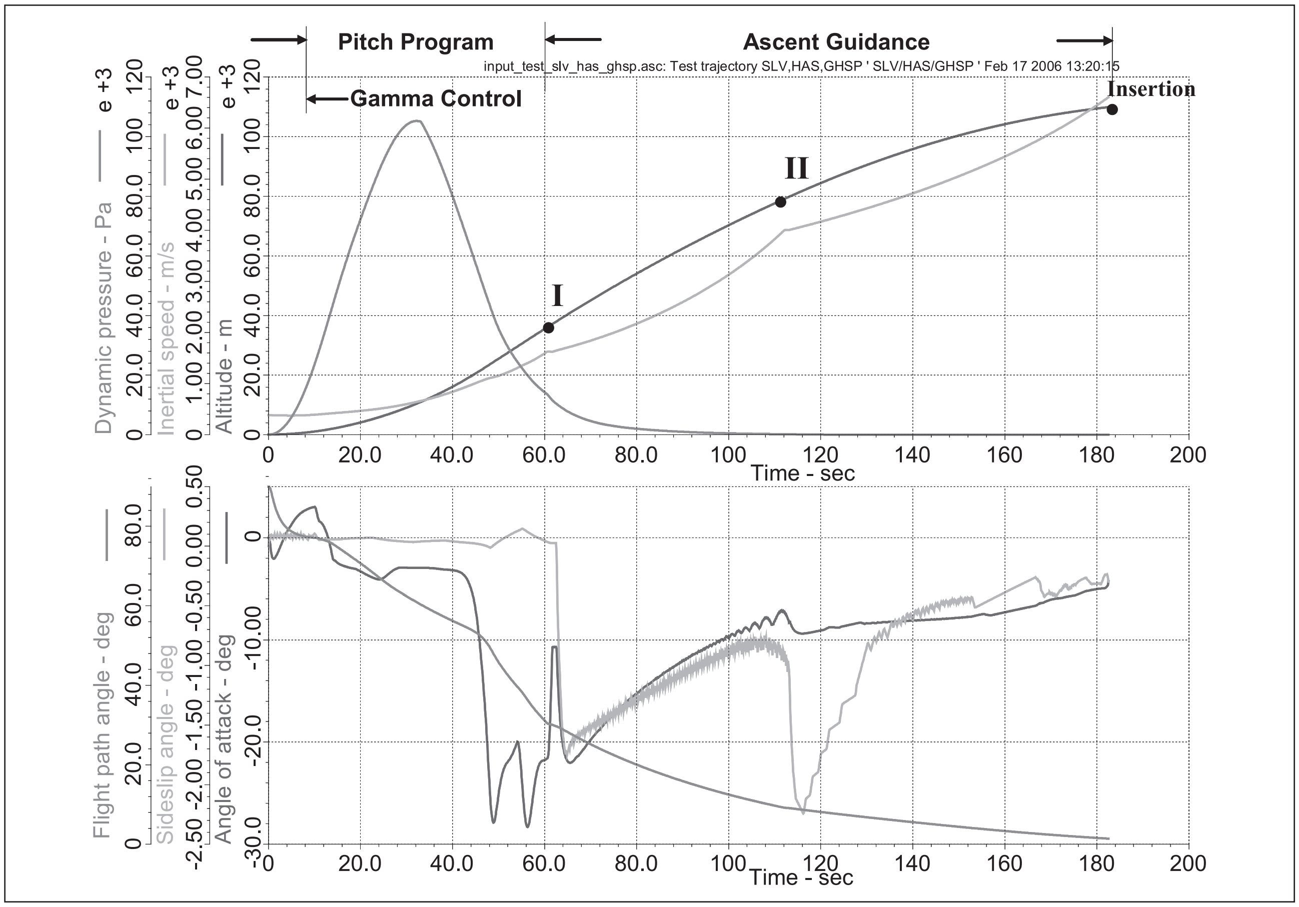

Figure 3. Trajectory parameters of the three stage booster ascent.

To evaluate the performance of the booster the engineer requires the traces of many trajectory parameters. Some of them are plotted with CADAC Studio 2Dim and displayed in Figure 3.

Note the dynamic pressure peak at 30 seconds into flight. The incidence angles are small during the peak to keep the structural loads within limits. Afterwards, the angle of attack increases dramatically to alter the flight path towards the insertion point.

This simulation makes use of the Round6 class common to all 6-DoF simulations over the WGS84 Earth. It uses the INS, GPS and star-track modules from other simulations. Only the aerodynamics(), propulsion(), guidance() and control() modules are specific to this application.

5.2 Generic Defense Missile Simulation

This simulation highlights the multiple instantiation of three vehicle objects. An aircraft launches a missile (blue) to intercept an attacking missile (red). The blue missile receives target updates during midcourse until its seeker locks onto the red missile for terminal homing.

The class structure has three branches. For the main missile object Cadac ← Flat6 ← Missile, for the red missile Cadac ← Flat3 ← Target, and for the aircraft Cadac ← Flat3 ← Aircraft. The modules are shown in Figure 4. Inside each object data flows through the protected arrays while across objects the data is provided by the communication bus ‘combus’ packets. Note that three levels of modeling fidelity are combined. The major focus of the analysis is on the blue missile. It is modeled in 6-DoF while the red missile is in 5-DoF and the aircraft in 3-DoF (though both, red missile and blue aircraft use the 3-DoF equations of motion).

The multiple instantiation of the vehicle objects is demonstrated in Figure 5. The aircraft launches two defensive missiles against two incoming threat missiles. The graph was drawn by the 3Dim plotting program of CADAC Studio.

As an example of a typical Monte Carlo result, Figure 6 shows the impact points of the blue missile on the red missile intercept plane created by 100 MC runs. The generation of this graph is fully automated in CADAC Studio. The CEP and the bivariate ellipse are shown. It reveals a

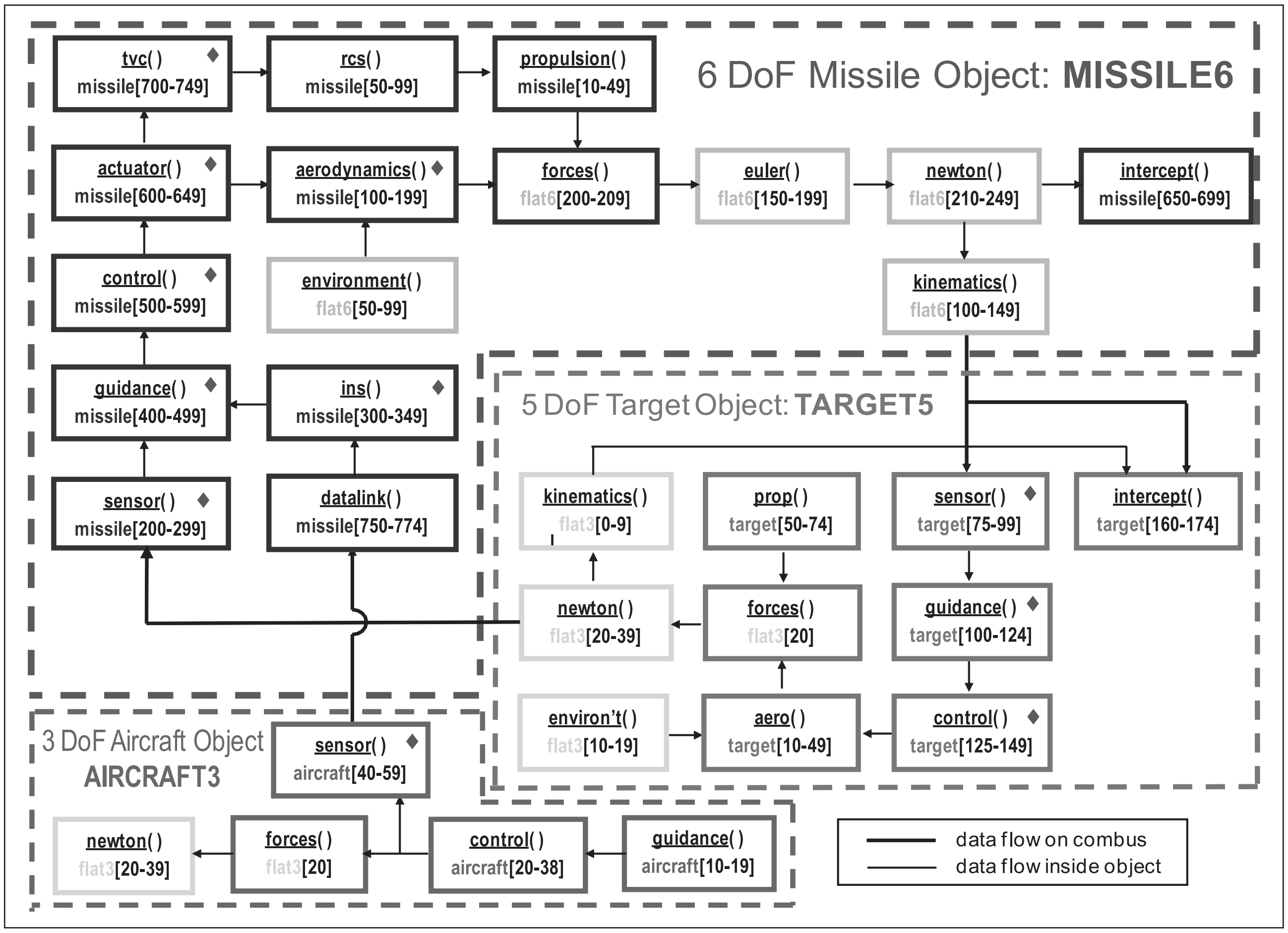
6 DoF Missile Object: MISSILE6
tvc( ) missile[700-749]
rcs( ) missile[50-99]
propulsion( ) missile[10-49]
actuator( ) missile[600-649]
aerodynamics( ) missile[100-199]
forces( ) flat6[200-209]
euler( ) flat6[150-199]
newton( ) flat6[210-249]
intercept( ) missile[650-699]
control( ) missile[500-599]
environment( ) flat6[50-99]
kinematics( ) flat6[100-149]
guidance( ) missile[400-499]
ins( ) missile[300-349]
sensor( ) missile[200-299]
datalink( ) missile[750-774]
5 DoF Target Object: TARGET5
kinematics( ) flat3[0-9]
prop( ) target[50-74]
sensor( ) target[75-99]
intercept( ) target[160-174]
newton( ) flat3[20-39]
forces( ) flat3[20]
guidance( ) target[100-124]
environ't( ) flat3[10-19]
aero( ) target[10-49]
control( ) target[125-149]
3 DoF Aircraft Object AIRCRAFT3
sensor( ) aircraft[40-59]
newton( ) flat3[20-39]
forces( ) flat3[20]
control( ) aircraft[20-38]
guidance( ) aircraft[10-19]
data flow on combus
data flow inside object

Figure 4. Modular architecture of the Generic Defense Missile simulation.

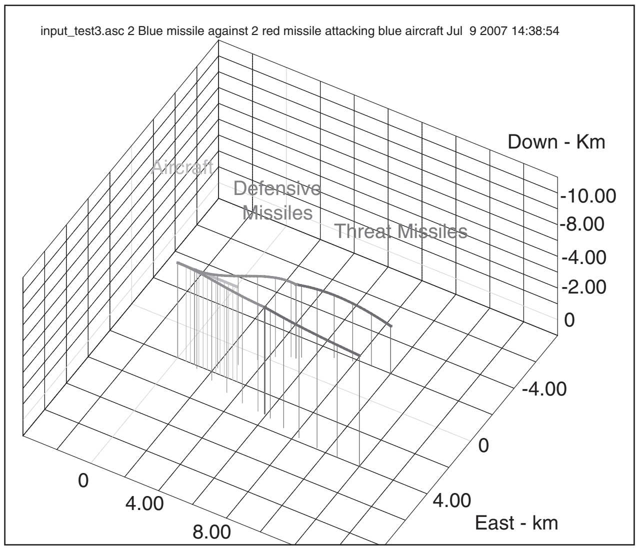

Figure 5. Two defensive missiles against two offensive missiles fly-out.

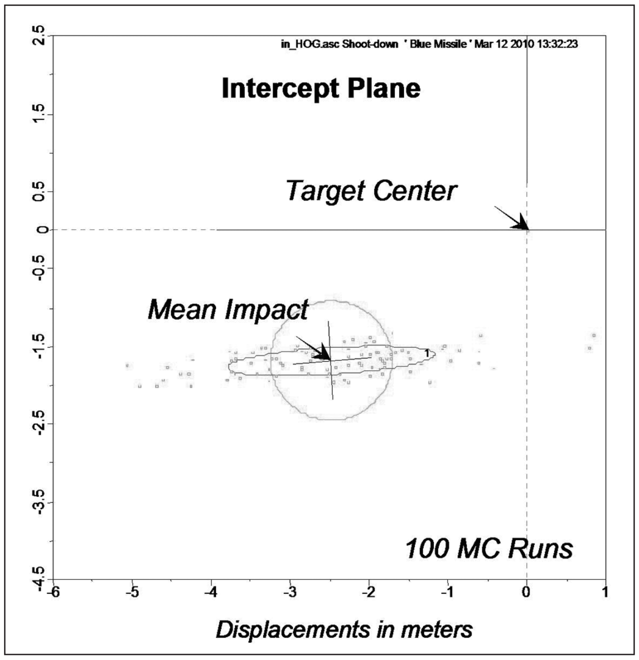

Figure 6. Impact of blue missile on red missile intercept plane.

significant bias in the system due to guidance and control lags and limiters.

The Generic Defense Missile simulation makes use of the common class Flat6 with its 6-DoF equations of motion over the flat Earth. Several modules of the blue missile are shared with other simulations. These include modules such as rcs(), tvc(), actuator(), ins() and sensor(). The modules aerodynamics(), propulsion(), guidance() and control() are specific to the Generic Defense Missile.

These are just two examples of the family of CADAC++ simulations shown in Table 1. Other simulations also demonstrate the commonality made possible by the class structure and modularity of the CADAC++ architecture.

6 Summary

The conversion of CADAC from FORTRAN to C++ is essentially complete. During the ten years since the switch has been made, several simulations were updated and new models created. CADAC Studio also experienced modifications to make it compatible with the C++ output. The new code has been applied to various US Air Force projects and found to be invaluable for concept explorations, technology assessments and mission-level studies.

References

1. Zipfel PH. Modeling and Simulation of Aerospace Vehicle Dynamics. 2nd ed. American Institute of Aeronautics and Astronautics, 2007.
2. Berndt JS. JSBSim, www.jsbsim.org. Last accessed: 19 Jan 2011.
3. Flightgear Version 2.0.0, http://www.flightgear.org (25 February 2010). Last accessed: 19 Jan 2011.
4. Hester J. CMD C++ Modeling Developer. Redstone Arsenal, AL: U.S. Army Research, Development and Engineering Command; jeffrey.hester@us.army.mil,
5. Zipfel PH. CADAC4, http://www.AIAA.org/content.cfm?pageid=403&ID=1592. Last accessed: 19 Jan 2011.
6. Zipfel PH. *Building Aerospace Simulations in C++*. 2nd ed. AIAA, 2008 (CD-ROM).
7. Zipfel PH. *Fundamentals of Six Degrees of Freedom Aerospace Simulations in FORTRAN and C++*. AIAA, 2004 (CD-ROM).
8. Zipfel PH. *Advanced Six Degrees of Freedom Aerospace Simulation and Analysis*. AIAA, 2005 (CD-ROM).
9. Ternion Corporation. FLAMES Flexible Analysis, Modeling, and Exercise System, http://www.ternion.com. Last accessed: 19 Jan 2011.
10. NIMA. *Department of Defense World Geodetic System 1984*. Report No.: TR 8350.2 4.Bethesda, MD: NIMA, July 1997.
11. NOAA. *U.S. Standard Atmosphere 1976*. Report No.: S/T 76-1562. U.S. Government Printing Office, 1976.
12. Dryden HL. A review of the statistical theory of turbulence. *Q Appl Math* 1943; 1: 7-42.
13. Bryson HO. *Applied Optimal Control*. Hemisphere Publishing Co., 1975.
14. Jaggers RF. *Multi-Stage Linear Tangent Guidance as Baseline for the Space Shuttle Vehicle*. Technical Report MSC-07458 (Internal Note MSC-EG-72-39). NASA, June 1972.